NoSQL

for Retail and eCommerce

Couchbase

TABLE OF CONTENTS

3 **Executive Summary**

4 **Retail and eCommerce companies embrace NoSQL to improve customer experience, efficiency and agility**

4 **Database performance and availability are key to delivering great customer experiences**

5 **NoSQL distributed databases provide better performance and availability than relational**

6 **NoSQL delivers faster, easier, more affordable scalability**

6 *Single-node type simplifies scaling*

6 *Bidirectional replication improves availability and data locality*

7 *Couchbase combines single node type and bidirectional replication*

7 **NoSQL drives increased operational efficiency: lower costs and faster time to market**

7 *Elastic scaling optimizes hardware usage*

8 *Data flexibility speeds innovation and time to market*

8 **NoSQL can support numerous retail and eCommerce use cases**

9 **NoSQL is well suited to many data types**

10 **Use Case Spotlight: Product Catalog**

10 *Product catalogs are challenging for relational databases*

10 **Product catalog data modeling in a NoSQL database**

12 **Retail and eCommerce data modeling in a NoSQL database**

13 **How to introduce a NoSQL database into a relational environment**

14 *NoSQL in a microservices architecture*

14 *Caching with NoSQL*

15 **As retail and eCommerce go mobile, NoSQL can deliver a better customer experience, online and in-store**

15 **Why Couchbase is a great NoSQL solution for retail and eCommerce**

16 **Conclusion: NoSQL is the right choice for many retail and eCommerce use cases**

Couchbase

NoSQL for Retail and eCommerce

Executive Summary

Digital Economy companies – including retail and eCommerce leaders like Walmart, Tesco, eBay, Fanatics, StubHub, Staples, cars.com and hundreds more – are embracing NoSQL database technology to build and run modern web and mobile applications. Why? Because NoSQL is better than relational technology in meeting the performance, scalability, availability, agility, and affordability requirements of these applications.

Rising customer expectations and competitive pressures are driving the need for NoSQL.

Rising customer expectations and competitive pressures are driving the need for NoSQL. Companies face increasing demands to deliver great customer experiences – i.e., fast, personalized, context- and location-aware. At the same time, they have to manage growing volumes of users and data, while reducing costs and time to market. These pressures create a new set of requirements for the operational database. It must:

- Service data requests with submillisecond latency
- Scale to meet peak demand (e.g. Black Friday), easily and affordably
- Provide 24x7x365 availability
- Easily accommodate evolving data types and queries
- Ingest, aggregate, and store multi-structured data from many sources
- Support multiple channels and devices
- Replicate data across data centers globally
- Integrate with other big data tools like Hadoop, Spark, Kafka and others
- Accelerate and simplify development

The operational database must also be versatile to support many use cases – e.g., customer profile management; shopping cart; session store; product and pricing catalog; 360-degree customer view and loyalty program management, to name just a few.

This paper provides an overview for CIOs, CTOs, architects, developers, and operations engineers interested in building and supporting modern applications on NoSQL:

- Why relational database technology (Oracle, SQL Server, MySQL, PostgreSQL, etc.) cannot meet the new requirements for web and mobile applications
- How NoSQL – specifically Couchbase, an open source document database – provides a better solution
- Examples of retail and eCommerce use cases supported by NoSQL, showing how data can be modeled and queried with Couchbase.

We conclude with a section on how to introduce NoSQL into a relational environment, emphasizing two approaches:

- NoSQL in a microservices architecture
- NoSQL for high-performance caching

Retail and eCommerce companies embrace NoSQL to improve customer experience, efficiency and agility

In 2016, 89% of companies expect to compete based on customer experience.

*(Source: **Gartner Survey**)*

In the Digital Economy, retail and eCommerce companies compete based on customer experience and operational efficiency – customers expect personalized engagements, relevant information, and instant gratification, but their expectations have to be met while reducing operational costs and time to market. In the Digital Economy, retail and eCommerce companies win by creating loyal customers who spend more, but cost less.

Companies not only face competition from within their industry – whether it's apparel, automotive, electronics, office supplies, or other products – but also from Internet companies like Amazon and eBay. These digital-born, tech-savvy companies are renowned for how they continuously improve their eCommerce applications, and for the speed and agility with which they release new features.

A recent CIO article on ways to improve the customer experience for online shoppers highlights the importance of site performance, customer reviews, stock availability, live chat, the checkout process, shopping carts, and more.

*(Source: **CIO**)*

For architects, developers, and operations engineers, the pressure to deliver great customer experiences, reduce costs, and accelerate innovation has never been greater. As a result, across all retail and eCommerce categories, from cars to clothes to office supplies and more, innovative companies – companies like cars.com, Fanatics, StubHub, Best Buy, Foot Locker, Nike, Staples, Tesco, Walmart, and hundreds more – are embracing NoSQL to improve customer experience, operational efficiency, and agility.

NoSQL is a modern database technology developed about a decade ago by leading Internet companies including Google, Facebook, Amazon and LinkedIn. It was created to overcome the limits of relational databases in meeting the requirements of modern web, mobile and IoT applications.

Database performance and availability are key to delivering great customer experiences

When it comes to customer experience, performance and availability are critical. Whether customers know it or not, they're interacting with a database. They're accessing product data, customer data, engagement data – and if the data is not readily available, the customer's experience suffers.

Walmart saw a sharp decline in their conversion rate when page load time increased from one to four seconds. However, for every one second of improvement, their conversion rate increased by 2%.

*(Source: **Walmart Labs**)*

Shoppers expect every request, whether it's finding and viewing a product, adding it to the shopping cart, or proceeding to checkout, to be handled immediately – not within seconds, but milliseconds. It doesn't matter what time it is, or what time zone they're in. It doesn't matter if it's Black Friday, Cyber Monday, or Super Sunday – shoppers expect eCommerce applications, web and mobile, to be available 24 hours a day, 365 days a year, anywhere in the world, and on any device.

Historically, relational databases have been the bottleneck, as DBAs and developers try to squeeze out every last drop of performance and work around the clock to maintain availability. However, with an ever-growing number of customers and rising expectations, they're fighting an uphill battle.

It's one reason why retail and eCommerce companies are embracing NoSQL: They require a database that's engineered for a higher level of performance and availability – capable of meeting the expectations of an online, 24/7, global customer base.

NoSQL distributed databases provide better performance and availability than relational

Applications with thousands, if not millions, of web and mobile customers can overwhelm relational databases with increasing, and more importantly, volatile workloads.

Typically, relational databases run on a single server, so the resources available to the database – processing, storage, and memory – are not only limited, they're fixed. This was not a problem when relational databases powered internal-facing applications with a limited number of users and predictable workloads. However, applications with thousands, if not millions, of web and mobile customers can overwhelm relational databases with increasing, and more importantly, volatile workloads.

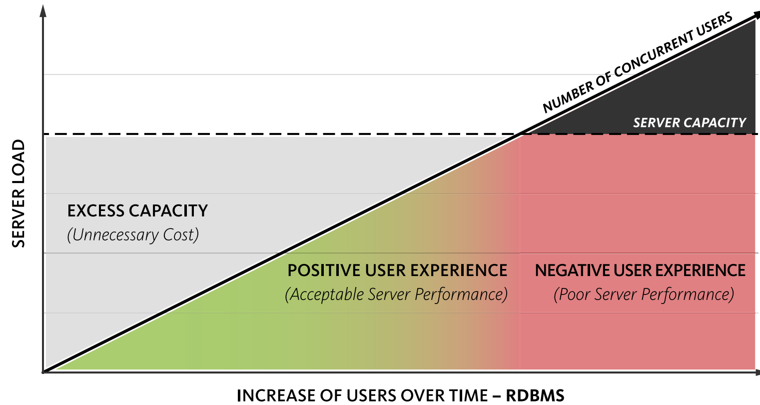

Figure 1: *Relational databases suffer from under/over-provisioning, leading to performance and cost issues.*

The only way to maintain performance as the number of customers and products grows, especially during peak periods, is to add more resources – often on demand. With relational databases, this is both difficult and costly, because you have to replace the RDBMS server with a bigger, more expensive server. But with many NoSQL databases, it's simply a matter of adding more servers to scale out. That's because they often run on a cluster of servers – i.e., they're distributed databases.

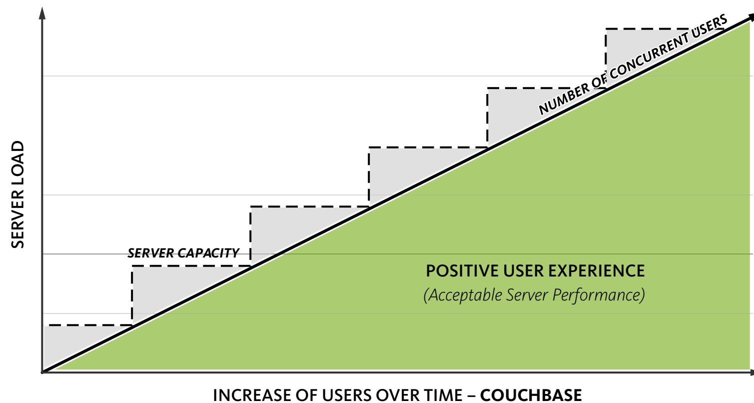

Figure 2: *NoSQL databases can increase capacity in small increments for more efficient use of hardware.*

The same is true for availability. If a relational database is running on a single server, and that server fails, the data becomes unavailable. NoSQL databases, when running on a cluster of servers, not only tolerate hardware failure, they expect it. That's why the data is replicated to multiple nodes. If one fails, the data remains available on the others.

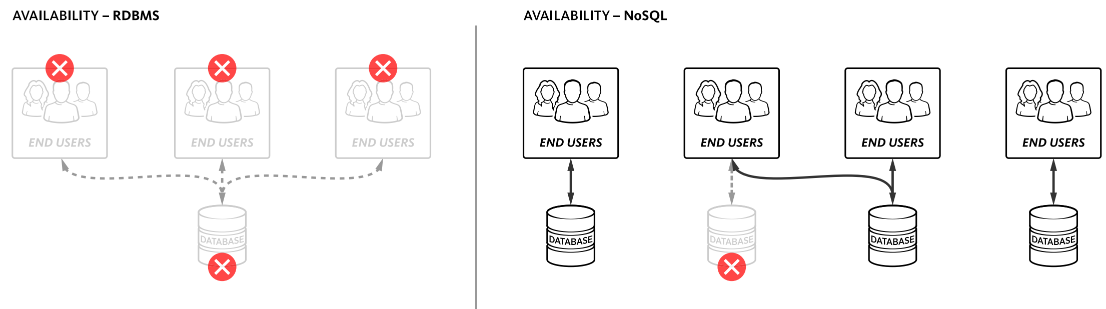

Figure 3: *Relational databases suffer from a single point of failure, but NoSQL databases do not.*

NoSQL delivers faster, easier, more affordable scalability

For a NoSQL database to effectively scale, it requires two things: a flat topology and bidirectional replication across data centers.

Single-node type simplifies scaling

It's not enough for a NoSQL database to be able to scale – it also has to scale on demand and with ease. This can be difficult for NoSQL databases that rely on complex topologies with multiple moving parts, like master-slave or other specialized node type topologies, for example. Such complexity introduces barriers to easy scaling, because it requires careful configuration of different nodes with different roles and different relationships.

Rather, every node should be capable of performing the same roles – a single node type. A flat topology with a single node type is easy to scale because it requires little more than adding a node to the cluster.

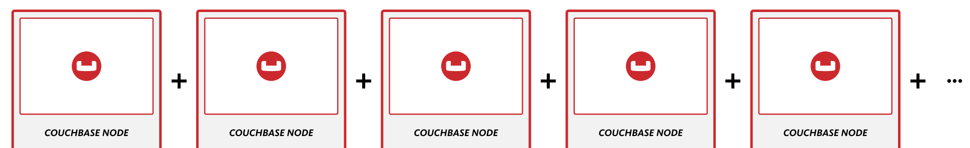

Figure 4: *It's easy to scale with a single-node architecture - start more nodes, add them to the cluster.*

Bidirectional replication improves availability and data locality

In order to provide the highest level of performance and availability to a geographically distributed customer base, the database must *also* be geographically distributed.

Bidirectional replication between data centers is also essential, because customers are often not confined to a single geography; they can be in different states, countries, or regions. In order to provide the highest level of performance and availability to a geographically distributed customer base, the database must also be geographically distributed.

However, being geographically distributed isn't enough: the database must be deployed in an active-active configuration using bidirectional replication. This enables eCommerce applications to read and write to the data center closest to the customer – providing local data access for the best possible performance. For example, the shopping carts of U.S. customers should be updated on a North American data center while the carts of UK customers should be updated on an EMEA data center. It would not make sense for all customer carts to be updated on a single, possibly geographically distant, higher latency, data center.

In addition, with active-active configurations, an entire data center can fail without interrupting availability. That's because applications can simply write to a different data center, or requests can be automatically rerouted by routers and load balancers. For example, if the North American data center failed, U.S. customers would access their online shopping carts from the EMEA data center – and they wouldn't even know.

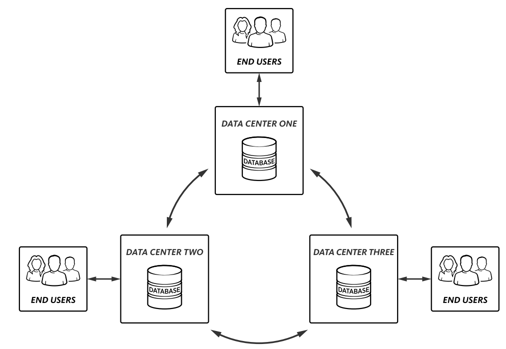

Figure 5: *Deploy to multiple data centers in an active/active configuration with bidirectional replication.*

Providing this kind of seamless failover without interrupting the customer experience is a challenge for NoSQL databases that rely on unidirectional replication (master-slave) between data centers. They can provide local access to mostly read-only data, like a product catalog, but they can't do it for things like shopping carts, customer profiles, and inventory – i.e., data services that may be location specific. In addition, because writes can only be handled by the data center with the primary node, applications have to wait for a new primary to be elected should the original fail – resulting in temporary loss of availability, potentially on a global scale.

Couchbase combines single node type and bidirectional replication

Among leading NoSQL databases, Couchbase stands out as the only one that combines a single-node architecture (i.e., flat topology) and bidirectional replication between data centers, to deliver fast and easy scalability, along with high performance and availability.

NoSQL drives increased operational efficiency: lower costs and faster time to market

In addition to improving the customer experience with greater performance and availability, NoSQL databases improve operational efficiency by reducing costs and enabling a faster time to market.

Elastic scaling optimizes hardware usage

Database workloads for retail and eCommerce companies can be highly volatile and unpredictable – spikes can occur due to events like Black Friday and Cyber Monday, as well as special promotions like flash sales or when an item goes viral. As a result, it can be challenging to maintain operational efficiency. When a relational database server is configured for peak workloads, and it's not operating under one, then it's over-provisioned.

Capacity can be adjusted simply by increasing or decreasing the number of nodes in a NoSQL database cluster, depending on workload.

Hardware, like any other resource, must be elastic – it must be possible to use more or less of it depending on demand. NoSQL databases can both scale out and scale in. Capacity can be adjusted simply by increasing or decreasing the number of nodes in a NoSQL database cluster, depending on workload. With a NoSQL database, it's possible to always use only as much hardware as needed – never too much, never too little. Because hardware is always optimized for the current workload, NoSQL is more cost efficient than relational.

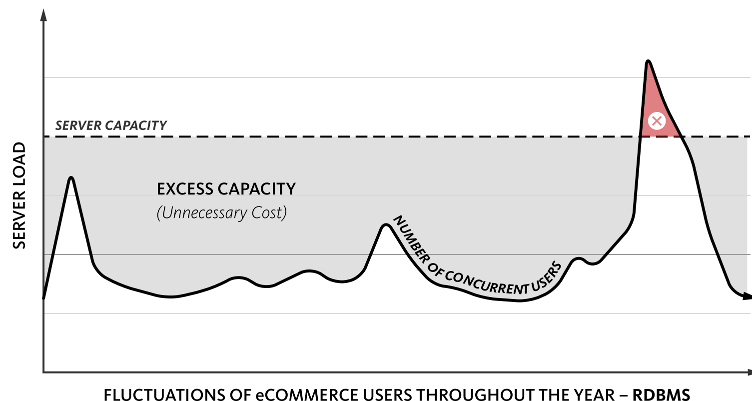

Figure 6: *Relational databases with fixed capacities cannot support by volatile workloads with high peaks.*

Data flexibility speeds innovation and time to market

Innovation is key to competitive success – for retail and eCommerce companies, that means a steady flow of new products, new offers, new programs, and more. And when it comes to innovation, time to market is critical. Relational databases, and wide-column stores, limit the pace of development due to their static schemas. Adding a new feature often requires changing the database schema, which in turn requires planning and coordination – both time-consuming tasks.

However, NoSQL databases – in particular key-value stores and document databases – leverage dynamic schemas. Rather than defining a static schema, they defer to the application. As a result, developers can create new features without having to wait for a schema change. When the database enables this kind of agile development instead of hindering it, new features can released far more quickly, resulting in faster time to market for retail and eCommerce companies.

NoSQL can support numerous retail and eCommerce use cases

Innovative retail and eCommerce companies are successfully leveraging NoSQL for use cases that involve products, customers, and interactions – for everything from product and pricing catalogs, to content management, to personalization to customer profiles, and more. All of these use cases benefit from one or more of the advantages of NoSQL: dynamic schema, low latency throughput, scalability and availability.

For example, when stored in a relational database, a product catalog requires a complex schema across multiple tables and complex queries to access it. However, when stored in a NoSQL document database, all of the information for a single product can be stored together in a single JSON document – making it much easier and faster to access.

Ingesting clickstream and session data to power real-time recommendation engines and create personalized experiences requires a database capable of sustaining high throughput while maintaining low latency.

When managing millions of customers, storing customer profiles and all of their interactions requires a database that can easily and affordably scale. It's a matter of volume, as well as velocity. Ingesting clickstream and session data to power real-time recommendation engines and create personalized experiences requires a database capable of sustaining high throughput while maintaining low latency.

Below are some of the common retail and eCommerce application use cases where companies are leveraging NoSQL for greater performance, availability, scalability, and agility:

Unified Customer Profile Management
Store and aggregate customer profile data to enable multiple services (e.g., authentication, personalization). Customer profiles include data that's local to the application/customer (demographic profile, behavior, history, location, preferences, etc.) as well as data aggregated from multiple internal and external sources and channels (financial profile, credit history, purchasing history, etc.), ultimately providing a 360-degree view of the customer.

Omnichannel Shopping Services
Provide customer-centric, personalized, low latency, scalable data services to support seamless, always-on shopping experiences for in-store, kiosk, mobile and web-based shopping. The shopping service must provide a personalized experience and be able to scale to handle peak internet and mobile device usage of the service.

Shopping Cart Session Management
Reliably manage real-time online shopping cart session data in memory for low latency performance. Allow customers to view and modify the contents of their online shopping cart, as well as finalize their purchases (ie, checkout) securely, quickly and easily.

Order & Fulfillment Management
Capture orders and generate appropriate fulfillment data in order to accelerate order processing, tracking and delivery to the customer. Applications typically include flexible business rules that manage how orders are fulfilled in order to minimize cost and enhance the customer experience. Amazon Prime and other premium fulfillment services are great examples of this customer and cost-driven fulfillment model.

Product Catalog Management
Scalably manage a unified repository of all product data, including product SKUs, descriptions (in multiple languages), images, ratings, reviews, etc. The product repository may also include product relationships or product "baskets" for products which are often purchased together, as well as competitive product information. Schema flexibility, rich data modeling abilities and performance are key requirements to provide enhanced product-customer interaction.

Price Catalog Management
Provide a low-latency repository of pricing data per product, including static as well as dynamic pricing rules (e.g., by geography, customer status, date & time, promotions, discounts, etc.) which need to be applied in order to generate a real-time, customer specific price. In retail and eCommerce applications pricing is never static – complex business rules drive optimized pricing. Customer expectations require that dynamic pricing applications return results in milliseconds.

Inventory & Availability Management
Manage and maintain inventory data to enable real-time inventory checks and updates. Inventory must be tracked both locally within a store, regionally, as well as globally, in order to support services such as in-store order fulfillment and lowest-cost, shortest-time delivery options. When availability falls below business-rule specified levels, the application should trigger actions in downstream legacy systems, such as Supply Chain Management (SCM) for example to trigger automatic re-stocking.

Personalization
Utilize market segmentation data, business rules and other relevant data in combination with the customer profile and product catalog repositories to feed a personalization / recommendation engine. Pre-generated as well as real-time recommendations are often combined in order to provide a seamless, highly responsive, personalized customer experience.

Loyalty Program Management
Aggregate and store relevant customer purchase history, loyalty status, and rewards activity (e.g., points, redemptions, discounts, promotions), and maintain associated flexible loyalty program business rules and policies. Design flexibility is a key requirement in this use case because each Loyalty Program is different. How loyal customers are rewarded is a key apect of long term customer retention for retail and eCommerce companies.

NoSQL is well suited to many data types

NoSQL document databases are well suited to many types of data – making them very effective as a general-purpose database for retail and eCommerce applications. They are ideal for product, clickstream, session, order, and user / member data – all part of today's customer experience, whether online or in-store.

NoSQL document databases are well suited to many types of data – making them very effective as a general-purpose database for retail and eCommerce applications.

BUSINESS CONCERN	PRODUCT			ENGAGEMENT				CUSTOMER	
USE CASE	CATALOG	CONTENT		PERSONALIZATION			TRANSACTIONS	PROFILES	
SERVICE	ITEMS			CLICKSTREAMS	SESSIONS		ORDERS	USERS	MEMBERS
DATA EXAMPLES	SKU / PRICE	DESCRIPTION	REVIEWS	RECOMMENDATIONS	HISTORY	LOCATION	PURCHASES	PREFERENCES	ACTIVITY
	INVENTORY	IMAGES	RATINGS	OFFERS / PROMOS	SHOPPING CART	CHECKOUT		WISH LIST	REWARDS
	COUPONS								

Figure 7: *The types of uses cases and data that retail / eCommerce companies use NoSQL databases for.*

Use Case Spotlight: Product Catalog

As a concrete example of how NoSQL can be a better fit for retail and eCommerce applications than relational databases, let's consider one of the most common use cases: **Product Catalog**.

Product catalogs are challenging for relational databases

Products can be very complex. In addition to a name and base price, a product can have significantly more data associated with it. Products can belong to multiple categories. Product data can include numerous features, specifications, descriptions and images. Data can include reviews, ratings, and other third-party generated content.

There are several ways to model product catalog data with a relational database. However, they're simply workarounds to get past limitations of relational modeling. No one approach is ideal. They all introduce new problems and challenges for developers. *(see table, right)*

DATA MODEL	PROBLEMS AND CHALLENGES
Entity Attribute Values	• Querying a product requires lots of joins • Querying a product requires hard-coded values • Adding a product requires multiple inserts • Indexes are inefficient due to low cardinality • Requires pivots, will not scale well
Class Table Inheritance	• Querying a product requires a join • Querying different products requires lots of joins • Adding a product requires two inserts • Adding a new product type requires schema changes
CLOB	• Can't create indexes to improve query performance • Have to read and write the entire value • Difficult to filter because the fields are not exposed

Product catalogs can easily be modeled with a NoSQL database

It's much easier to model product catalogs with a NoSQL document database, because all data for a single product can be stored together. It can all be stored in a single document instead of multiple rows, often in multiple tables. Not only is it easier to model the data, it's simpler and faster to access – there's no need to perform a query with multiple joins or to pivot the results.

It's much easier to model product catalogs with a NoSQL document database, because all data for a single product can be stored together.

Product (Book)

```
key:
products::books::chasm_city

document:
{
  “type”: “book”,
  “title”: “Chasm City”,
  “author”: “Alastair Reynolds”,
  “skus”: [
    {
      “sku”: 123456,
      “isbn-13”: “9780441010646”,
      “format”: “paperback”,
      “pages”: 704
      “published”: 2003
      “price”: 9.99},
    {
      “sku”: 234567,
      “isbn-13”: “9780575068773”,
      “format”: “hardcover”,
      “pages”: 528
      “published”: 2001
      “price”: 19.99}]
  “reviews”: {
    “average”: 4.4,
    “count”: 26},
  “categories”: [
    “Science Fiction and Fantasy”,
    “Science Fiction”,
    “Space Opera”]
}
```

Product (Movie)

```
key:
products::movies::2001_a_space_odyssey

document:
{
  “type”: “movie”,
  “title”: “2001: A Space Odyssey”,
  “director”: “Stanley Kubrick”,
  “released”: 1968
  “skus”: [
    {
      “sku”: 789012,
      “format”: “blu-ray”,
      “duration”: 149,
      “price”: 9.99,
      “reviews”: {
        “average”: 4.7,
        “count”: 150}},
    {
      “sku”: 890123,
      “format”: “laserdisc”,
      “duration”: 149,
      “price”: 124.99,
      “reviews”: {
        “average”: 4.5,
        “count”: 130}}]
  “categories”: [
    “Science Fiction”]
}
```

Uniquely with Couchbase, product data can be queried via a SQL-based query language, N1QL ("nickel"), or via simple key-value operations.

Uniquely with Couchbase, product data can be queried via a SQL-based query language, **N1QL** ("nickel"), or via simple key-value operations. In the following examples, the results are displayed as tables for simplicity. However, within applications, the results can be JSON documents, providing much more flexibility – especially when querying multiple product types.

Example One: List all products by title and price

```
SELECT p.title, p.type, s.format, s.price FROM products p UNNEST p.skus s
ORDER BY p.title, s.price
```

Title	*Type*	*Format*	*Price*
2001: A Space Odyssey	Movie	Blu-Ray	9.99
2001: A Space Odyssey	Movie	Laserdisc	124.99
Chasm City	Book	Paperback	9.99
Chasm City	Book	Hardcover	19.99

Example Two: Find all movies under $10

```
SELECT p.title, p.director, s.format, s.price FROM products p UNNEST p.skus s
WHERE p.type = "movie" AND s.price < 10
```

Title	*Director*	*Format*	*Price*
2001: A Space Odyssey	Stanley Kubrick	Blu-Ray	9.99

Example Three: Find all laserdisc movies

```
SELECT p.title, p.director, s.format, s.price FROM products p UNNEST p.skus s
WHERE p.type = "author" AND s.format = "laserdisc"
```

Title	*Director*	*Format*	*Price*
2001: A Space Odyssey	Stanley Kubrick	Laserdisc	124.99

Example Four: Find all science fiction books

```
SELECT p.title, p.author, s.format, s.price FROM products p UNNEST p.skus s
WHERE p.type = "book" AND ANY cat IN p.categories SATISFIES cat = "Science Fiction" END
```

Title	*Author*	*Format*	*Price*
Chasm City	Alastair Reynolds	Paperback	9.99
Chasm City	Alastair Reynolds	Hardcover	19.99

All sorts of retail and eCommerce data can easily be modeled in a NoSQL database

It's easy to model other types of data with NoSQL databases, too – shopping carts, product reviews, inventory, and more. As with product catalogs, this data is easier and faster to access with a NoSQL database.

Document databases provide a great deal of flexibility by supporting nested elements like arrays and objects.

In addition, document databases provide a great deal of flexibility by supporting nested elements like arrays and objects. For example, product reviews – they could be embedded within the product document, stored as individual documents, or stored as a group of documents. In the example below, ten product reviews are stored per document using an array – making it easy and fast to paginate through them, ten at a time.

Shopping Cart

```
key:
cart::18484071

document:
{
  "type": "cart",
  "customer": 101
  "items": [
    {
      "sku": 123456,
      "title": "Chasm City",
      "price": 9.99
      "quantity": 1},
    {
      "sku": 789012,
      "title": "2001: A Space Odyssey",
      "price": 19.99,
      "quantity": 1}]
}
```

Product Reviews

```
key:
reviews::123456::10

document:
{
  "type": "reviews",
  "sku": 123456,
  "reviews": [
    {
      "customer": 101,
      "quality": 5.0,
      "value": 4.5,
      "overall": 5.0,
      "feedback": "This thing is awesome."},
    {
      customer": 102,
      "quality": 4.0,
      "value": 4.0,
      "overall": 4.0,
     "feedback": "This thing is pretty good."},
    ... ]
}
```

The following examples highlight the **subdocument API** in Couchbase Server. It enables applications to read or write specific fields within a document for maximum performance. This includes updating fields, incrementing counters, adding elements to arrays, and more.

Example One: Add two copies of Chasm City to the cart instead of one

```
bucket.mutateIn("cart::18484071").replace("items[0].quantity, 2).doMutate();
```

Example Two: Add a copy of Neuromancer to the cart

```
JsonObject item =
  JsonObject.create().put("sku", "024680").put("title", "Neuromancer").put("price", 9.99).put("quantity", 1);

bucket.mutateIn("cart::18484071").pushBack("items", item).doMutate();
```

Example Three: Add a review to Chasm City

```
JsonObject review = JsonObject.create();

review.put("customer", "1").put("quality", 5.0).put("value", 5.0).put("overall", 5.0).put("feedback", "I loved it!");

// this is the 12th review, add it to the document with reviews 11-20
bucket.mutateIn("products::books::chasm_city::reviews::11-20").pushFront("reviews", review);
```

Example Four: Get the first ten reviews of 2001: A Space Odyssey on Blu-ray

```
bucket.get("products::movies::sku::789012::reviews::1-10");
```

Modeling inventory data

In addition to product catalogs, modeling **inventory data** with a document database is easy and flexible, too. Below are two examples. The first models inventory by product, the second by instance. For example, when there is a fixed number of instances (e.g. *event tickets*), you can store each instance as a separate document.

Inventory (by SKU)

```
key:
inventory::123456

document:
{
  "type": "inventory",
  "sku": "123456"
  "stores": [
    {
      "store": 100,
      "city": "San Jose",
      "state": "California",
      "quantity": 250},
    {
      "store": 101,
      "city": "Mountain View",
      "state": "California",
      "quantity": 120}]
}
```

Inventory (by Item)

```
key:
inventory::tickets::superbowl:12-100

document:
{
  "type": "ticket",
  "row": 12
  "seat": 100
  "status": "available"
}

inventory::tickets::superbowl:12-102

{
  "type": "ticket",
  "row": 12
  "seat": 102
  "status": "sold"
}
```

Example One: Update the inventory, 100 copies of Chasm City (softcover) have arrived

```
bucket.mutateIn("inventory::123456").counter("quantity", 100L).doMutate();
```

Example Two: Change the inventory status, set 12-102 for the Super Bowl has been purchased

```
bucket.mutateIn("inventory::tickets::superbowl:12-100").replace("status", "sold").doMutate();
```

How to introduce a NoSQL database into a relational environment

Most innovative retail and eCommerce companies have embraced NoSQL by successfully introducing it into their relational environments.

NoSQL document databases are excellent general-purpose databases for retail and eCommerce applications. However, that doesn't mean there isn't a place for relational databases. In particular, relational databases are well suited to legacy business management applications, such as enterprise resource planning (ERP) and supply chain management (SCM). Whereas these are employee-facing applications, NoSQL is better suited to meet the performance, scalability, availability, and agility requirements of interactive customer-facing applications.

In fact, the most innovative retail and eCommerce companies have embraced NoSQL by successfully introducing it into their relational environments. It's a matter of architecture, and there are two popular approaches to adding NoSQL into an existing relational database environment: microservices and caching.

NoSQL in a microservices architecture

Most successful retail and eCommerce applications are deployed using a microservices architecture. In a microservices architecture, applications are deployed as a set of independent, full-stack services.

An important concept of microservices architectures is that of decentralized data management. Where a monolithic application would store all data in a single database, a set of microservices can and should store their own data in their own databases. This often leads to polyglot persistence – different services using different types of databases. While some services may continue to use a relational database, others may use NoSQL databases – some may use a document database, others may use a graph database.

With a microservices architecture, eCommerce companies can choose when and where to introduce a NoSQL database, enabling them to migrate to NoSQL one service at a time.

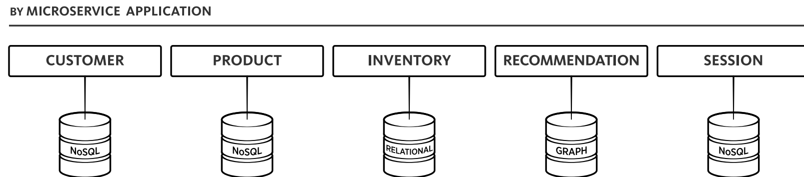

Figure 8: *A microservices architecture supports different services using different types of databases.*

With a microservices architecture, eCommerce companies can choose **when and where** to introduce a NoSQL database, enabling them to migrate to NoSQL one service at a time.

Caching with NoSQL

In some environments, it may be better to augment an existing relational database with a NoSQL database. In this architecture, data is loaded into a NoSQL database that is deployed between the application and the relational database. In a sense, the NoSQL database functions like a cache. For example, the product catalog can be exported from the relational database to the NoSQL database. When the data changes in the relational database – for example, the addition of new products – the NoSQL database is updated.

However, the data doesn't have to be read-only – inventory data, for example. The relational database may be the primary source of record, but retail and eCommerce applications will interact with the NoSQL database for faster performance. While it's possible for the inventory to become out of date for a brief time – until the relational and NoSQL databases are synchronized – the benefits of better performance, and thus a better customer experience, far outweigh the drawbacks.

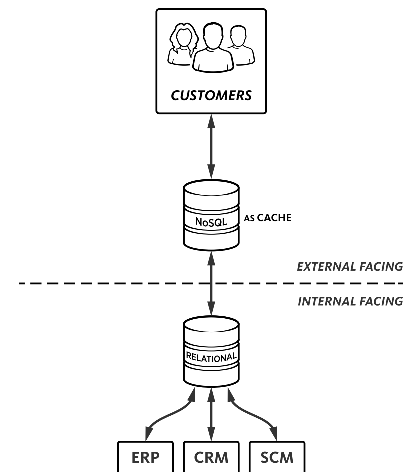

Figure 9: *Deploy a NoSQL database on top of a relational database to improve performance and reduce costs.*

Applications can leverage Database Change Protocol (DCP) in Couchbase Server to tap into a change stream, and apply these changes to the underlying relational database. In the reverse direction, with Oracle Database for example, applications can use Oracle GoldenGate to capture changes and apply them to Couchbase Server. Finally, applications can use the Couchbase Kakfa Connector to stream changes between Couchbase Server and a relational database in both directions.

In addition to providing better performance and availability, adding a NoSQL database can reduce the costs associated with maintaining relational databases. For example, a NoSQL database can be deployed to lessen the load on a mainframe – reducing MIPS and thus costs – or to avoid having to deploy the relational database on bigger hardware, and thereby avoid higher licensing and maintenance costs.

As retail and eCommerce go mobile, NoSQL can deliver a better customer experience, online and in-store

Every year, more and more customers are shopping online via mobile devices. According to a BI Intelligence report, **20% of all eCommerce will be mobile in 2016**, 45% by 2020 (up to $284 billion in sales). However, many retail and eCommerce companies struggle with the mobile experience, leading to lower conversion rates.

To create a mobile experience that is on par, or exceeds, the web experience requires a native mobile solution. A key component of that solution is an embedded database for mobile devices, providing mobile apps with local data access. It not only improves performance, because the data is local – it also improves availability, because an embedded database ensures the app will always work whether or not there's an active network connection. Furthermore, the solution must provide built-in synchronization – enabling the database to push data to the consumer, and vice-versa.

A native mobile solution can be used to improve both online and in-store experiences. For example, relevant coupons and rewards can be pushed to customers' mobile phones while they're shopping – even using beacons to detect what department they're in to increase relevancy and thus conversion. Customers can add items to a virtual shopping cart or browse their wishlist – perhaps receive a notification when something is in stock or on sale. By leveraging the full power of a native mobile solution, there are many possibilities for improving the customer experience, online or in-store.

Couchbase Mobile is the only native, NoSQL-based solution for mobile apps.

Couchbase Mobile is the only native, NoSQL-based solution for mobile apps. It is comprised of Couchbase Lite and Couchbase Sync Gateway, and when paired with Couchbase Server, it's a complete platform for mobile apps. In addition, it's a cross-platform solution that is available for iOS, Android, .NET/Java, and Mac OS X.

Why Couchbase is a great NoSQL solution for retail and eCommerce

Couchbase is a powerful NoSQL solution for a broad range of use cases, delivering the high performance, scalability, availability and agility that today's retail and eCommerce applications require. Numerous retail and eCommerce companies – including Walmart, eBay, Office Depot, Tesco, Fanatics, Nu Skin, and many more – have chosen Couchbase for several key advantages.

Memory-centric architecture
Couchbase takes full advantage of all available memory to give your application the sub-millisecond responsiveness that today's shoppers expect.

Integrated cache
While other NoSQL databases like MongoDB require a third-party cache – adding to both cost and complexity – Couchbase has a fully integrated cache that delivers blazing performance. No need for a separate product to install and manage.

Powerful, SQL-based query language
Unique among all NoSQL document databases, Couchbase provides N1QL ("nickel") – a powerful query language that lets developers easily query JSON data using familiar, SQL-like expressions.

Built-in high availability and disaster recovery
Couchbase comes with high availability within a cluster and provides market-leading cross datacenter replication (XDCR) capabilities to support DR and data locality requirements. No need for complicated third-party systems. You have full control over the topology – unidirectional, bidirectional or any configuration you need.

Complete, GUI-based admin console
Among NoSQL document databases, only Couchbase provides a fully integrated GUI-based management console, complete with hundreds of pre-built metrics and easy to use tools like push-button scaling, rebalancing, and memory tuning.

Always-on mobile support
Couchbase Mobile is a complete NoSQL solution for mobile application support. It includes Couchbase Lite – an embedded JSON database for devices – and Sync Gateway, a pre-built solution that syncs the device with the cloud. Couchbase Mobile lets you easily support use cases such as in-store personalized apps, point of sales systems, and mobile-optimized digital catalogs.

Conclusion: NoSQL is the right choice for many retail and eCommerce use cases

If you're running into scalability, performance, or availability challenges with your relational database – and you're looking to speed development and innovation – it's probably time to consider NoSQL technology.

If you're running into scalability, performance, or availability challenges with your relational database – and you're looking to speed development and innovation – it's probably time to consider NoSQL technology.

Hundreds of leading retail and eCommerce companies have already made the move, deploying NoSQL to support a growing number of use cases from customer profile management personalization, to product catalogs, shopping carts, 360-degree customer view, and many more.

Getting started is easy

Download the software **and install it in a non-production corner of your IT environment.** Couchbase Server is available for Microsoft Windows, Apple OS X, and leading Linux platforms such as Red Hat, Debian, and CentOS.

Explore our sample travel reservation app. A ready-made testing environment is waiting for your input, or your can install it on your own machines and get intimate with the app's source code.

Talk to a Couchbase Solutions Engineer **about your retail or eCommerce use case.** We have dedicated sales and support offices in North America, Europe, and Asia Pacific, and trusted partners all over the globe.

About Couchbase

Couchbase delivers the world's highest performing NoSQL distributed database platform. Developers around the world use the Couchbase platform to build enterprise web, mobile, and IoT applications that support massive data volumes in real time. The Couchbase platform includes Couchbase Server, Couchbase Lite - the first mobile NoSQL database, and Couchbase Sync Gateway. Couchbase is designed for global deployments, with configurable cross data center replication to increase data locality and availability. All Couchbase products are open source projects. Couchbase customers include industry leaders like AOL, AT&T, Bally's, Beats Music, BSkyB, Cisco, Comcast, Concur, Disney, eBay, KDDI, Nordstorm, Neiman Marcus, Orbitz, PayPal, Rakuten / Viber, Tencent, Verizon, Wells Fargo, Willis Group, as well as hundreds of other household names. Couchbase investors include Accel Partners, Adams Street Partners, Ignition Partners, Mayfield Fund, North Bridge Venture Partners, and West Summit.

2440 West El Camino Real | Ste 600
Mountain View, California 94040

1-650-417-7500

www.couchbase.com